实验要求

语法分析程序，实现对算术表达式的语法分析。要求所分析算数表达式由如下的文法产生：

$$
\begin{aligned}
&E \rightarrow E+T|E-T|T \\
&T \rightarrow T*F|T/F|F \\
&F \rightarrow (E)|\mathrm{num}
\end{aligned}
$$

要求在对输入的算术表达式进行分析的过程中，依次输出所采用的产生式。

程序设计说明

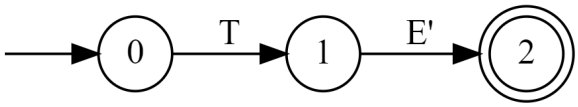
递归下降方法

消去左递归

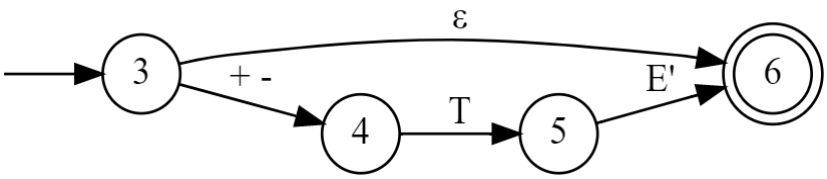
$$
\begin{aligned}
E &\to TE' \\
E' &\to +TE' \mid -TE' \mid \varepsilon \\
T &\to FT' \\
T' &\to *FT' \mid /FT' \mid \varepsilon \\
F &\to (E) \mid \mathrm{num}
\end{aligned}
$$

预测分析程序状态转换图

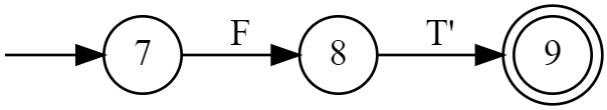
E:



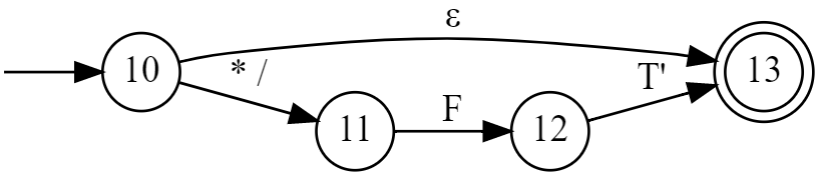
E':



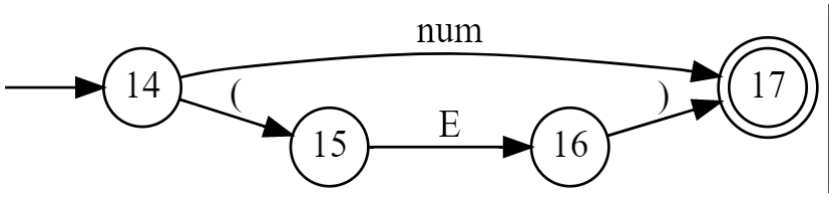
T:



T':

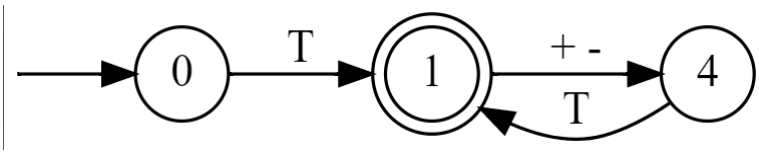


F:

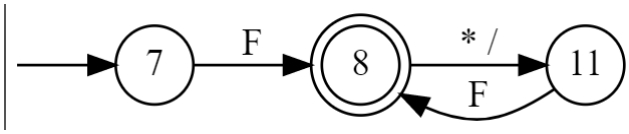


化简后得到：

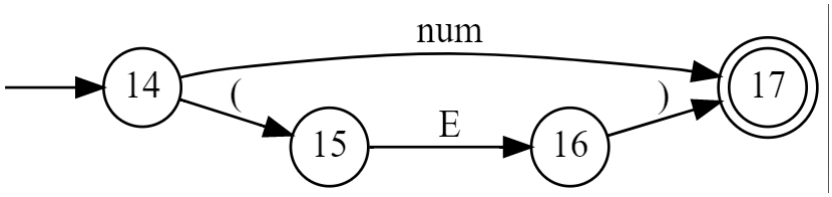
E:



T:



F:



根据状态转换图，就可以书写代码了。

代码的输入是使用词法分析分析得到的符号序列。

```
#include <bits/stdc++.h>

#include "lex.hpp"

using Lex::TokenSequence;

struct Recursive {
  TokenSequence tokens; // 当前在分析的符号流

  auto Current() const { // 当前分析的第一个符号
    return tokens.front();
  }

  void ProcE() { // E的递归分析程序
    std::cerr << "In ProcE\n";
    ProcT();
    if (Current().second == "+" || Current().second == "-") {
      tokens.pop();
      ProcT();
    }
  }

  void ProcT() { // T的递归分析程序
    std::cerr << "In ProcT\n";
    ProcF();
    if (Current().second == "*" || Current().second == "/") {
      tokens.pop();
      ProcF();
    }
  }

  void ProcF() { // F的递归分析程序
    std::cerr << "In ProcF\n";
    if (Current().first == TOKEN::FLOAT || Current().first == TOKEN::INT) {
      tokens.pop();
    } else if (Current().second == "(") {
      tokens.pop();
      ProcE();
      if (Current().second == ")")
        tokens.pop();
      else
        Error();
    } else
      Error();
  }

  void Error() {
    std::cerr << "Error\n";
```

```
    exit(0);
  }

  void Run(TokenSequence tok) {
    tokens = tok;
    tokens.push({TOKEN::BLANK, ""});
    ProcE();
    if (tokens.size() != 1)
      Error();
    else
      std::cerr << "Success\n";
  }
};

int main(int argc, const char **argv) {
  if (argc != 2)
    std::cerr << "Usage: ./recursive-descent source_file\n";
  else {
    std::ifstream src(argv[1]);
    Recursive r;
    r.Run(Lex::clangLex.GenSequence(src));
  }
  return 0;
}
```

LL(1)分析

代码结构

定义Token类型如下：

```
struct Token {       // 文法符号的定义
  bool T;            // T=1 为终止符
  std::string str;   // 符号的名称
  bool operator<(Token const &x) const {
    return T != x.T ? T < x.T : str < x.str;
  }
  bool operator==(Token const &x) const { return T == x.T && str == x.str; }
  bool operator!=(Token const &x) const { return T != x.T || str != x.str; }
};
```

定义一些相关的类型简写如下：

```
using Expr = std::vector<Token>;  // 生成式右部，定义成Token序列
template <class T>
using SMap = std::map<std::string, T>;
using SSet =
    std::set<std::string>;  // 为了方便，将和string有关的set和map重命名一下
using Grammar = SMap<
    std::vector<Expr>>;  // 文法定义成从左部到包含着所有可能的右部的map，比如A
                         // → B|C，就会被存储为[A]→{{B},{C}}
```

由于LL(1)和LR(1)分析均需要求First集合，因此我为LL(1)自动机和LR(1)自动机设置了一个共同的基类，内含两类自动机都需要的函数。

```
struct GrammarAutomataBase {  // 语法分析自动机的基类
  const std::string init;     // 初始状态
  const Grammar g;            // 语法
  std::set<Token> allTokens;  // 所有符号的集合
  SSet eps;                   // 能转移到空的非终止符集合
  SMap<SSet> first;           // first集合

  GrammarAutomataBase;

  SSet SequenceFirst;

  void Epsilon();             // 求出能转移到epsilon的非终结符

  void First();               // 求First集合
};
```

对于LL(1)自动机的类，剩余部分的结构如下：

```
struct LL1Automata : GrammarAutomataBase {
  SMap<SSet> follow;                        // follow集合
  SMap<SMap<std::vector<const Expr *>>> M;  // 分析表

  LL1Automata(std::string init, Grammar g);

  void Follow();           // 构造Follow集合

  void MTable();   // 构造分析表

  void Print();

  void Run(Lex::TokenSequence tokens);
};
```

FIRST集合构造

回顾课件上的构造方法：

- 若 $X \to a\ldots$，则把 a加入到$\mathrm{FIRST}(X)$中。
- 若 $X \to \varepsilon$ ，则 ε 也加入到$\mathrm{FIRST}(X)$中。
- 若 $X \to Y\ldots$是产生式，则把$\mathrm{FIRST}(Y)$中的所有非ε元素加入到$\mathrm{FIRST}(X)$中。
- 若 $X \to Y_1Y_2\ldots Y_k$是产生式，且对某个i，有 $\varepsilon \in \mathrm{FIRST}(Y_1) \cap \mathrm{FIRST}(Y_2) \cap \ldots \cap \mathrm{FIRST}(Y_{i-1})$，则把$\mathrm{FIRST}(Y_i)$中的所有非$\varepsilon$元素加入到$\mathrm{FIRST}(X)$中。若对所有$Y_i$有$\varepsilon \in \mathrm{FIRST}(Y_i)$，则将$\varepsilon$加入$\mathrm{FIRST}(X)$。

因此，问题的求解可以被分成两个步骤：

1. 求出所有能到达ε的状态，由此就能得到每个集合的FIRST集合是由那些集合并集得到。
2. 按照得到的集合生成规则，求出每个集合的正确结果。

下面来分步看这个问题。

求到达ε的状态

将所有的包含$X \to \varepsilon$生成式的符号X标记为可达。

对于每个生成式$X \to Y_1 Y_2 \dots Y_n$，判断当前的所有Y_i是否已经被标记。如果是的话，就可以把X也标记上。

如果没有出现新的被标记的点，就结束程序。

```
  void Epsilon() {              // 求出能转移到epsilon的非终结符
    std::queue<std::string> Q;  // 能转移到epsilon的非终结符队列
    SMap<std::vector<std::pair<std::string, const Expr *>>>
        possible;  // 对于每个非终结符，维护所有右部包含它的生成式
    for (auto &[a, exprs] : g) {
      allTokens.insert({0, a});
      for (auto &expr : exprs) {
        if (expr.empty()) {
          eps.insert(a);
          Q.push(a);  // 将所有能转移到空的符号插入队列
        } else if (std::count_if(expr.begin(), expr.end(),
                                 [](auto const &x) -> bool { return x.T; }) ==
                   0)
          for (auto &i : expr)
            possible[i.str].push_back(
                {a,
                 &expr});  // 若一个生成式的右部只包含非终结符，则每个非终结符能转移到空都
可能给整个式子转移到空带来一线生机
        for (auto &i : expr) allTokens.insert(i);
      }
    }
    while (!Q.empty()) {
      auto s = Q.front();
      Q.pop();
      for (auto &[a, expr] : possible[s])
        if (!eps.count(a))
          if (!std::count_if(expr->begin(), expr->end(),
                             [&](auto const &sub) -> bool {
                               return sub.T || !eps.count(sub.str);
                             })) {  // 若生成式右部的所有符号都能转移到空了
            eps.insert(a);
            Q.push(a);
          }
      possible[s].erase(
          std::remove_if(
              possible[s].begin(), possible[s].end(),
              [&](auto const &x) -> bool { return eps.count(x.first); }),
          possible[s].end());  // 删掉所有整体能变成空的生成式，避免冗余判断
    }
  }
```

生成每个集合

得到所有能到达ε的状态之后，就能知道每个集合是由哪些集合直接并集得到的。这个并集的关系是一个传递关系，若$A \subset B, B \subset C$，则$A \subset C$，可以用求解传递闭包的Warshall算法求解。

```
  void First() {                                        // 求First集合
    std::map<Token, std::map<Token, bool>> include;  // First集合的包含关系
```

```
    for (auto &i : allTokens) include[i][i] = 1;
    for (auto &[a, exprs] : g) {
      for (auto &expr : exprs) {
        for (auto &i : expr) {
          include[{0, a}][i] =
              1;  // 生成式右部的每个符号，如果它前面的部分能转移到空，那么生成式左部的First
就包含了它的First
          if (i.T || !eps.count(i.str)) break;
        }
      }
    }

    for (auto &k : allTokens)
      for (auto &i : allTokens)
        for (auto &j : allTokens)
          include[i][j] |= include[i][k] & include[k][j];  // Warshall求传递闭包

    for (auto &i : allTokens)
      if (!i.T) {
        for (auto &j : allTokens)
          if (j.T && include[i][j]) first[i.str].insert(j.str);
        if (eps.count(i.str)) first[i.str].insert(EPSILON.str);
      }
  }
```

FOLLOW集合构造

回顾课件上的构造方法：

- 对文法开始符号S，置$于$\mathrm{FOLLOW}(S)$中，$为输入符号串的右尾标志。
- 若$A \to \alpha B\beta$是产生式，则把$\mathrm{FIRST}(\beta)$中的所有非ε元素加入到$\mathrm{FOLLOW}(B)$中。
- 若$A \to \alpha B$是产生式，或 $A \to \alpha B\beta$ 是产生式并且$\beta \Rightarrow \varepsilon$，则把$\mathrm{FOLLOW}(A)$中的所有元素加入到$\mathrm{FOLLOW}(B)$中。

由此，在求出所有能推出ε的符号之后，每个符号的FOLLOW集合也可以写成若干FIRST集合和若干FOLLOW集合并集的形式，也可以用闭包算法来求解。

```
  void Follow() {           // 构造Follow集合
    allTokens.insert(END);  // 符号集合中加入表示结尾的符号$
    std::map<Token, std::map<Token, bool>> include;  // 表示Follow集合的包含关系
    include[{0, init}][END] = 1;
    for (auto &[a, exprs] : g) {
      for (auto &expr : exprs) {
        std::set<std::string> last =
            {};  // 对于每个右部Y1Y2 ... Yn，从右到左枚举每一项，维护可能接在Yi后面的终止符
        bool epsSuffix = 1;  // 表示当前枚举到的后缀是否可以转移到空
        for (int i = expr.size() - 1; i >= 0; --i) {
          if (!expr[i]
                   .T) {  // 对于非终止符，将所有last中的符号加入它的follow集合
            for (auto &s : last) include[expr[i]][{1, s}] = 1;
            if (epsSuffix)
              include[expr[i]][{0, a}] =
                  1;  // 若后缀整体可以为空，那么它的Follow集合还包含了生成式左部的Follow
集合
          }
          if (expr[i].T || !eps.count(expr[i].str))
            last =
```

```
                {};  // 若出现了转移不到空的非终止符或者终止符，那么当前的last中的符号就不能接到更靠左的符号后面了
          if (expr[i].T)  // 前面的符号后方可以接上终止符自身
            last.insert(expr[i].str);
          else  // 前面的符号可以接上当前符号First集合中所有非空项
            for (auto &s : first[expr[i].str])
              if (s != EPSILON.str) last.insert(s);
          epsSuffix &=
              !expr[i].T &&
              eps.count(
                  expr[i]
                      .str);  //如果当前符号不能转移到空了，那后缀就无法转移到空了
        }
      }
    }
    for (auto &k : allTokens)
      for (auto &i : allTokens)
        for (auto &j : allTokens)
          include[i][j] |= include[i][k] & include[k][j];  // Warshall
    for (auto &i : allTokens)
      if (!i.T) {
        for (auto &j : allTokens)
          if (j.T && include[i][j]) follow[i.str].insert(j.str);
      }
  }
```

M表构造

课件中给出了如下的伪代码

```
for (文法G的每个产生式A→α) {
    for (每个终结符号a∈FIRST(α)) {
            把 A→α 放入M[A,a]中;
        if (ε∈FIRST(α))
            for (任何b∈FOLLOW(A))
                把 A→α放入M[A,b]中;
    }
for (所有无定义的M[A, a])
    标上错误标志;
```

直接简单翻译一下就行了。

```
  void MTable() {  // 构造分析表
    for (auto &[a, exprs] : g)
      for (auto &expr : exprs)
        for (auto &s : SequenceFirst(expr.begin(), expr.end()))
          if (s ==
              EPSILON
                  .str)  // 右部能转移到空，则对左部follow集合中的元素b，把该生成式加入M[左部,b]中
            for (auto &b : follow[a]) M[a][b].push_back(&expr);
          else  // 对First(右部)中的非空元素s，把该生成式加入M[左部,s]中
            M[a][s].push_back(&expr);
  }
```

分析过程

用栈存储当前待推导的符号，用队列存储输入的符号序列。

若栈顶为终结符，则期望队首是同样的终结符。那么就可把这个终结符从两个数据结构中弹出。

若栈顶为非终结符，则期望存在唯一的$M[\text{栈顶}][\text{队首}]$的推导表项，就可以按照这个规则将栈顶的符号替换成生成式的右部符号。注意，右部符号需要逆序压入栈。

```
void Run(Lex::TokenSequence tokens) {
  std::stack<Token> S;
  S.push(END);  // 栈底放一个$
  tokens.push({TOKEN::BLANK, END.str});
  S.push({0, init});  // 从起始状态开始推导
  TableContent app = {{{"Stack"}, {"Input"}, {"Strategy"}, {"Prefix"}}};
  std::string grammar;  // 当前分析出的语法串的前缀
  auto Error = [&]() {
    std::cout << "LL(1) Parsing Failed.\nPartial Parsing Sequence:\n";
    PrintTable(app, {30, 30, 30, 30});
    exit(0);
  };
  while (!tokens.empty()) {
    std::string expr;
    app.push_back(
        {{StringfyContainer(S, [](auto x) -> std::string { return x.str; })},
         {StringfyContainer(
             tokens, [](auto x) -> std::string { return x.second; }, 1)}});
    if (S.top().T) {  // 栈顶为终结符
      if (S.top().str ==
          tokens.front().second) {  // 弹出状态栈栈顶和符号队列队首
        if (!S.top().T || S.top() != END) grammar += S.top().str + ' ';
        tokens.pop();
        S.pop();
      } else
        Error();
    } else {
      auto T =
          M[S.top().str]
           [tokens.front().second];  // 栈顶为非终结符，则按照相应的生成式规约
      if (T.size() != 1)
        Error();
      else {
        auto &e = T.front();
        expr = StringfyExpr(S.top().str, *e);
        S.pop();
        for (int i = e->size() - 1; i >= 0; --i)
          S.push(e->at(i));  // 将生成式右部的符号逆序插入栈中
      }
    }
    app.back().push_back({expr});
    app.back().push_back({grammar});
  }
  if (!S.empty())
    Error();
  else {
    std::cout << "LL(1) Parsing Complete.\nFull Parsing Sequence:\n";
    PrintTable(app, {30, 30, 30, 30});
  }
```

```
  }
```

LR(1)分析

代码结构

定义分析表中项目和项目集如下：

```
struct Item {  // 语法分析项目
  std::string a;
  const Expr *expr;
  std::string n;
  int cur;
  // [a→expr(with . before cursor), n]

  bool operator<(Item const &x) const {
    if (a != x.a) return a < x.a;
    if (n != x.n) return n < x.n;
    if (cur != x.cur) return cur < x.cur;
    return expr < x.expr;
  }
};
```

除去基类之外剩下部分的定义如下：

```
using ISet = std::set<Item>;  // 项目集

struct LR1Automata : GrammarAutomataBase {
  SMap<SSet> follow;
  std::set<ISet> I;   //项目集规范族
  const ISet *start;  // 包含了 [S'→.S, $] 的项目集
  std::string
      realInit;  // 由于输入是拓广文法，真实的初始状态是初始状态唯一的生成式的右部
  std::map<const ISet *, SMap<const ISet *>> goTo;
  std::map<const ISet *,
           SMap<std::tuple<std::string, const Expr *, const ISet *>>>
      action;  // action中的每个表项式一个<a,e,s>的tuple，若e和s均为NULL，表示ACC；若e为
NULL，则表示shift
               // s，否则表示reduce a → e
  std::map<const ISet *, int> id;  // 为了方便输出，给每个项目集定一个编号

  LR1Automata(std::string init, Grammar g);

  void Table();   // 生成分析表

  void CalcI();

  ISet Closure(ISet S);

  ISet Go(ISet S, Token X);

  std::string StringfyAction(
      std::tuple<std::string, const Expr *, const ISet *> p);
```

```
  void Print();

  void Run(Lex::TokenSequence &tokens);
};
```

Closure求法

课件中介绍如下：

1. I中的每一个项目都$\in \text{Closure}(I)$。
2. 若项目 $[A \to \alpha \cdot B\beta, a] \in \text{Closure}(I)$，且$B \to \eta$是G的一个产生式，则对任意$b \in \text{First}(\beta a)$，$[B \to \cdot\eta, b] \in \text{Closure}(I)$。

由此，可以书写如下代码：

```
  ISet Closure(ISet S) {
    ISet S0;
    do {
      S0 = S;
      for (auto i : S0) {
        if (i.cur != i.expr->size() && !i.expr->at(i.cur).T) {
          Expr backward(i.expr->begin() + i.cur + 1, i.expr->end());
          backward.push_back({1, i.n});
          for (auto &n : SequenceFirst(
                   backward.begin(),
                   backward.end())) {  // 处理后缀符号串的First，把相应表项加入
            for (auto const &expr : g.at(i.expr->at(i.cur).str))
              S.insert({i.expr->at(i.cur).str, &expr, n, 0});
          }
        }
      }
    } while (S.size() > S0.size());
    return S;
  }
```

构造项目集规范族

有伪代码

```
C={closure( { [S'→·S, $] } ) };
do
    for (C中的每一个项目集I和每一个文法符号X)
        if (go(I, X)不为空，且不在C中)
            把 go(I, X) 加入C中;
while (没有新项目集加入C中);
```

实现也是比较简单的：

```
  void CalcI() {
    I = {{Closure({{init, &g.at(init).front(), END.str, 0}})}};
    start = &(*I.begin());
    std::set<ISet> I0;
    do {
      I0 = I;
      for (auto &s : I)
        for (auto &t : allTokens) {
```

```
            auto gos = Go(s, t);
            if (!gos.empty()) I.insert(gos);
        }
    } while (I.size() > I0.size());  // 重复直到项目集规范族不再变化
    int no = 0;
    for (auto &k : I) id[&k] = no++;
}
```

构造分析表

构造出项目集规范族之后，对于每个项目集I_i：

1. 若$[A \to \alpha \cdot a\beta, b] \in I_i$，且$\text{go}(I_i, a) = I_j$，则置$\text{action}[i, a] = \text{shift } j$
2. 若$[A \to \alpha \cdot, a] \in I_i]$，且$A \neq S'$，则置$\text{action}[i, a] = \text{reduce } A \to \alpha$
3. 若$[S' \to S \cdot, \$] \in I_i$，则置$\text{action}[i, \$] = \text{ACC}$
4. 若对非终结符号A，有$\text{go}(I_i, A) = I_j$，则置$\text{goto}[i, A] = j$

```
void Table() {  // 生成分析表
    for (auto &S : I) {
        for (auto &i : S) {
            if (i.cur == i.expr->size()) {  // 形如 A -> B. 的项目
                if (i.a != init)
                    action[&S][i.n] = {i.a, i.expr, NULL};  // 正常的reduce
                else {
                    assert(i.expr->size() == 1 &&
                           (!i.expr->front().T && i.expr->front().str == realInit) &&
                           i.n == "$");
                    action[&S][i.n] = {
                        "", NULL, NULL};  // [S -> S. , $]的状态，得到$之后的action为ACC
                }
            } else {  // 形如 A -> BC.D 的项目
                auto t = i.expr->at(i.cur);
                assert(I.count(Go(S, t)));
                auto gop = &(*I.find(Go(S, t)));  //正常向后转移一步
                if (t.T)
                    action[&S][t.str] = {"", NULL,
                                         gop};  // 若下一个符号为终结符，则为shift
                else
                    goTo[&S][t.str] = gop;  //否则在goto中填上到的位置，用于规约后的转移
            }
        }
    }
}
```

规约过程

初始的状态为$S' \to \cdot S$所在的状态。

使用一个栈来记录当前的经过的状态，用另一个栈记录当前走过的前缀。同样还是用队列来记录输入的符号。

若$\text{action}[\text{状态栈顶}][\text{队首}]$无定义，则报错。

否则，若当前action为shift，则将输入的队首压入符号栈并且弹出，同时把shift到的状态压入状态栈。

否则，为reduce动作。此时，前缀栈顶应当是正序存储着reduce规约式的右部符号们。从前缀栈中将它们全部弹出，同时同步地从状态栈中弹出相同个数个元素。假设当前规约式的左部符号为A，则将A压入前缀栈中，并且将$\text{goto}[\text{状态栈顶}][A]$压入状态栈中。

代码如下：

```
void Run(Lex::TokenSequence &tokens) {
  tokens.push({TOKEN::BLANK, END.str});
  std::stack<Token> prefix;          // 语法前缀栈
  std::stack<const ISet *> states;   // 状态栈
  states.push(start);                // 从[S'→.S,$]所在项目集开始

  TableContent t = {{{"States"}, {{"Prefix"}}, {{"Input"}}, {{"Strategy"}}}};
  while (!tokens.empty()) {
    auto Error = [&]() {
      std::cout << "LR(1) Parsing Failed.\nPartial Parsing Sequence:\n";
      PrintTable(t, {25});
      exit(0);
    };
    if (!action[states.top()].count(tokens.front().second))
      Error();
    else {
      auto [a, expr, s] = action[states.top()][tokens.front().second];
      t.push_back(
          {{StringfyContainer(
               states,
               [&](auto x) → std::string { return std::to_string(id[x]); })},
           {StringfyContainer(prefix,
                              [](auto x) → std::string { return x.str; })},
           {StringfyContainer(
               tokens, [](auto x) → std::string { return x.second; }, 1)},
           {StringfyAction({a, expr, s})}});
      if (!expr && !s) {  // ACC 如果不出问题的话，此时tokens应该空了
        tokens.pop();
      } else if (s) {  // shift
        prefix.push({1, tokens.front().second});
        tokens.pop();
        states.push(s);
      } else {  // reduce
        for (int r = expr→size(); r--; prefix.pop(), states.pop())
          assert(prefix.top() == expr→at(r));
        prefix.push({0, a});
        assert(goTo[states.top()].count(prefix.top().str));
        states.push(goTo[states.top()][prefix.top().str]);
      }
    }
  }
  std::cout << "LR(1) Parsing Complete.\nFull Parsing Sequence:\n";
  PrintTable(t, {25});
}
```

Lex-Yacc

使用了Linux环境下的`flex`做词法分析，使用`bison`生成语法分析器。

flex

由于在本次作业中只需要分析包含数字和`+ - * / ( )`的输入，因此我没有直接用上次的Lex程序，而是写了一个更简单的。

```
%{
#include "stdio.h"
#include "y.tab.h"
%}

%%
(([1-9])([0-9])*)|(0x[0-9A-Z]+)|(0[0-7]*))(U|L|UL|LL|ULL)?)|(([1-9][0-9]*)\.
[0-9]+(e[+|-]?[1-9][0-9]*)?)|[1-9][0-9]*e[+|-]?[1-9][0-9]*)(L)?) { printf("num =
%s\n",yytext);return NUM; }
[-/+*()\n] { printf("mark = '%s'\n", yytext[0]=='\n'?"\\n":yytext);return
yytext[0]; }
. { printf("other\n");return 0; }
%%

int yywrap(void) {
  return 1;
}
```

在这个文件中，我引用了`y.tab.h`。这个文件会被`bison`生成出来，并且包含`NUM`这个符号的定义，供词法分析程序和语法分析程序使用。代码中的`yytext`存储了当前识别的符号串的内容，可以直接输出使用。

使用命令`flex lex.l`，可以生成`lex.yy.c`。

bison

配套的语法分析程序如下：

```
%{
#include <stdio.h>
void yyerror(const char *s) {
  printf("error %s\n",s);
}
%}

%token NUM

%%
S : E '\n'  { printf("success\n"); }
  ;
E : E '+' T
  | E '-' T
  | T
  ;
T : T '*' F
  | T '/' F
  | F
  ;
F : '(' E ')'
  | NUM
  ;
```

```
%%

int main(int argc,const char **argv) {
  freopen(argv[1],"r",stdin);
  return yyparse();
}
```

其中`%%`包含的部分为文法的生成式。由于表达式后面一定要有一个分隔符，否则无法区分表达式结束，因此我把`\n`作为分隔符，识别表达式+`\n`的串。当成功识别到之后，输出`success`。

`%token NUM`语句表示声明了`NUM`这样一个符号，如上所述，这个符号会在`yy.tab.h`里生成出来，被flex程序引用之后，会在`yylex()`的结果中出现。

使用命令`bison -vtdy yacc.y`，会生成`y.output` `y.tab.c` `y.tab.h`。其中，`y.output`描述了对文法分析得到的`LALR(1)`分析方法，内容大致如下：

```
Grammar

    0 $accept: S $end

    1 S: E '\n'
...

Terminals, with rules where they appear

$end (0) 0
'\n' (10) 1
'(' (40) 8
')' (41) 8
...


Nonterminals, with rules where they appear

$accept (11)
    on left: 0
S (12)
    on left: 1, on right: 0
...


State 0

    0 $accept: . S $end

    NUM  shift, and go to state 1
    '('  shift, and go to state 2

    S  go to state 3
    E  go to state 4
    T  go to state 5
    F  go to state 6


State 1

    9 F: NUM .
```

```
    $default  reduce using rule 9 (F)


State 2

    8 F: '(' . E ')'

    NUM  shift, and go to state 1
    '('  shift, and go to state 2

    E  go to state 7
    T  go to state 5
    F  go to state 6
...
```

最后，使用`gcc -o cc y.tab.c lex.yy.c`，可以生成名为`cc`的可执行文件。

测试报告

可以使用`build.sh`编译所需的文件。

递归下降

测试1

```
输入文件:
(3-2)**(4/(10+5))

运行: ./recursive-descent source 输出
In ProcE
In ProcT
In ProcF
In ProcE
In ProcT
In ProcF
In ProcT
In ProcF
In ProcF
Error
```

测试2

```
输入文件:
(3-2)*(4/(10+5))


运行: ./recursive-descent source 输出
In ProcE
In ProcT
In ProcF
```

```
In ProcE
In ProcT
In ProcF
In ProcT
In ProcF
In ProcF
In ProcE
In ProcT
In ProcF
In ProcF
In ProcE
In ProcT
In ProcF
In ProcT
In ProcF
Success
```

LL(1)

grammar1文件中存储了消去左递归的文法

```
E
E → T E'
E' → + T E' | - T E' | eps
T → F T'
T' → * F T' | / F T' | eps
F → ( E ) | num
```

测试1

输入文件:
(3-2)*(4/10)

运行: ./ll1 grammar1 source > ll1.out 输出到文件中

```
FIRST and FOLLOW set:
+----+---------------+------------------+
|    |FIRST          |FOLLOW            |
+----+---------------+------------------+
|E   |{(,num }       |{$ ,) }           |
+----+---------------+------------------+
|E'  |{+,-,eps }     |{$ ,) }           |
+----+---------------+------------------+
|F   |{(,num }       |{$ ,) ,* ,+ ,- ,/ }|
+----+---------------+------------------+
|T   |{(,num }       |{$ ,) ,+ ,- }     |
+----+---------------+------------------+
|T'  |{*,/,eps }     |{$ ,) ,+ ,- }     |
+----+---------------+------------------+
M table:
+----+---------+---------+---------+---------+---------+---------+---------+
|    |$        |(        |)        |*        |+        |-        |/        |num      |
+----+---------+---------+---------+---------+---------+---------+---------+---------+
|E   |         |E -> TE' |         |         |         |         |         |E -> TE' |
+----+---------+---------+---------+---------+---------+---------+---------+---------+
|E'  |E' -> eps|         |E' -> eps|         |E' -> +TE'|E' -> -TE'|         |         |
+----+---------+---------+---------+---------+---------+---------+---------+---------+
|F   |         |F -> (E) |         |         |         |         |         |F -> num |
+----+---------+---------+---------+---------+---------+---------+---------+---------+
|T   |         |T -> FT' |         |         |         |         |         |T -> FT' |
+----+---------+---------+---------+---------+---------+---------+---------+---------+
|T'  |T' -> eps|         |T' -> eps|T' -> *FT'|T' -> eps|T' -> eps|T' -> /FT'|         |
+----+---------+---------+---------+---------+---------+---------+---------+---------+
LL(1) Parsing Complete.
Full Parsing Sequence:
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|Stack                              |Input                              |Strategy               |Prefix                             |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E                                |( num - num ) * ( num / num ) $    |E -> TE'               |                                   |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T                             |( num - num ) * ( num / num ) $    |T -> FT'               |                                   |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' F                          |( num - num ) * ( num / num ) $    |F -> (E)               |                                   |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' ) E (                      |( num - num ) * ( num / num ) $    |                       |(                                  |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' ) E                        |num - num ) * ( num / num ) $      |E -> TE'               |(                                  |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' ) E' T                     |num - num ) * ( num / num ) $      |T -> FT'               |(                                  |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' ) E' T' F                  |num - num ) * ( num / num ) $      |F -> num               |(                                  |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' ) E' T' num                |num - num ) * ( num / num ) $      |                       |( num                              |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' ) E' T'                    |- num ) * ( num / num ) $          |T' -> eps              |( num                              |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' ) E'                       |- num ) * ( num / num ) $          |E' -> -TE'             |( num                              |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' ) E' T -                   |- num ) * ( num / num ) $          |                       |( num -                            |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' ) E' T                     |num ) * ( num / num ) $            |T -> FT'               |( num -                            |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' ) E' T' F                  |num ) * ( num / num ) $            |F -> num               |( num -                            |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' ) E' T' num                |num ) * ( num / num ) $            |                       |( num - num                        |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' ) E' T'                    |) * ( num / num ) $                |T' -> eps              |( num - num                        |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' ) E'                       |) * ( num / num ) $                |E' -> eps              |( num - num                        |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' )                          |) * ( num / num ) $                |                       |( num - num )                      |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T'                            |* ( num / num ) $                  |T' -> *FT'             |( num - num )                      |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' F *                        |* ( num / num ) $                  |                       |( num - num ) *                    |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' F                          |( num / num ) $                    |F -> (E)               |( num - num ) *                    |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' ) E (                      |( num / num ) $                    |                       |( num - num ) * (                  |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' ) E                        |num / num ) $                      |E -> TE'               |( num - num ) * (                  |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' ) E' T                     |num / num ) $                      |T -> FT'               |( num - num ) * (                  |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' ) E' T' F                  |num / num ) $                      |F -> num               |( num - num ) * (                  |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' ) E' T' num                |num / num ) $                      |                       |( num - num ) * ( num              |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' ) E' T'                    |/ num ) $                          |T' -> /FT'             |( num - num ) * ( num              |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' ) E' T' F /                |/ num ) $                          |                       |( num - num ) * ( num /            |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' ) E' T' F                  |num ) $                            |F -> num               |( num - num ) * ( num /            |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' ) E' T' num                |num ) $                            |                       |( num - num ) * ( num / num        |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' ) E' T'                    |) $                                |T' -> eps              |( num - num ) * ( num / num        |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' ) E'                       |) $                                |E' -> eps              |( num - num ) * ( num / num        |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T' )                          |) $                                |                       |( num - num ) * ( num / num )      |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E' T'                            |$                                  |T' -> eps              |( num - num ) * ( num / num )      |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$ E'                               |$                                  |E' -> eps              |( num - num ) * ( num / num )      |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
|$                                  |$                                  |                       |( num - num ) * ( num / num )      |
+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------+
```

测试2

输入文件:
(3-2)*(4/10)

运行: ./ll1 grammar1 source > ll1.out 输出到文件中
（省略文法部分）

```
LL(1) Parsing Failed.
Partial Parsing Sequence:
+-------------------------+-------------------------+-------------------------+-------------------------+
|Stack                    |Input                    |Strategy                 |Prefix                   |
+-------------------------+-------------------------+-------------------------+-------------------------+
|$ E                      |( num - num ) * ( num /*10) $ |E -> TE'                 |                         |
+-------------------------+-------------------------+-------------------------+-------------------------+
|$ E' T                   |( num - num ) * ( num /*10) $ |T -> FT'                 |                         |
+-------------------------+-------------------------+-------------------------+-------------------------+
|$ E' T' F                |( num - num ) * ( num /*10) $ |F -> (E)                 |                         |
+-------------------------+-------------------------+-------------------------+-------------------------+
|$ E' T' ) E (            |( num - num ) * ( num /*10) $ |                         |(                        |
+-------------------------+-------------------------+-------------------------+-------------------------+
|$ E' T' ) E              |num - num ) * ( num /*10) $   |E -> TE'                 |(                        |
+-------------------------+-------------------------+-------------------------+-------------------------+
|$ E' T' ) E' T           |num - num ) * ( num /*10) $   |T -> FT'                 |(                        |
+-------------------------+-------------------------+-------------------------+-------------------------+
|$ E' T' ) E' T' F        |num - num ) * ( num /*10) $   |F -> num                 |(                        |
+-------------------------+-------------------------+-------------------------+-------------------------+
|$ E' T' ) E' T' num      |num - num ) * ( num /*10) $   |                         |( num                    |
+-------------------------+-------------------------+-------------------------+-------------------------+
|$ E' T' ) E' T'          |- num ) * ( num /*10) $       |T' -> eps                |( num                    |
+-------------------------+-------------------------+-------------------------+-------------------------+
|$ E' T' ) E'             |- num ) * ( num /*10) $       |E' -> -TE'               |( num                    |
+-------------------------+-------------------------+-------------------------+-------------------------+
|$ E' T' ) E' T -         |- num ) * ( num /*10) $       |                         |( num -                  |
+-------------------------+-------------------------+-------------------------+-------------------------+
|$ E' T' ) E' T           |num ) * ( num /*10) $         |T -> FT'                 |( num -                  |
+-------------------------+-------------------------+-------------------------+-------------------------+
|$ E' T' ) E' T' F        |num ) * ( num /*10) $         |F -> num                 |( num -                  |
+-------------------------+-------------------------+-------------------------+-------------------------+
|$ E' T' ) E' T' num      |num ) * ( num /*10) $         |                         |( num - num              |
+-------------------------+-------------------------+-------------------------+-------------------------+
|$ E' T' ) E' T'          |) * ( num /*10) $             |T' -> eps                |( num - num              |
+-------------------------+-------------------------+-------------------------+-------------------------+
|$ E' T' ) E'             |) * ( num /*10) $             |E' -> eps                |( num - num              |
+-------------------------+-------------------------+-------------------------+-------------------------+
|$ E' T' )                |) * ( num /*10) $             |                         |( num - num )            |
+-------------------------+-------------------------+-------------------------+-------------------------+
|$ E' T'                  |* ( num /*10) $               |T' -> *FT'               |( num - num )            |
+-------------------------+-------------------------+-------------------------+-------------------------+
|$ E' T' F *              |* ( num /*10) $               |                         |( num - num ) *          |
+-------------------------+-------------------------+-------------------------+-------------------------+
|$ E' T' F                |( num /*10) $                 |F -> (E)                 |( num - num ) *          |
+-------------------------+-------------------------+-------------------------+-------------------------+
|$ E' T' ) E (            |( num /*10) $                 |                         |( num - num ) * (        |
+-------------------------+-------------------------+-------------------------+-------------------------+
|$ E' T' ) E              |num /*10) $                   |E -> TE'                 |( num - num ) * (        |
+-------------------------+-------------------------+-------------------------+-------------------------+
|$ E' T' ) E' T           |num /*10) $                   |T -> FT'                 |( num - num ) * (        |
+-------------------------+-------------------------+-------------------------+-------------------------+
|$ E' T' ) E' T' F        |num /*10) $                   |F -> num                 |( num - num ) * (        |
+-------------------------+-------------------------+-------------------------+-------------------------+
|$ E' T' ) E' T' num      |num /*10) $                   |                         |( num - num ) * ( num    |
+-------------------------+-------------------------+-------------------------+-------------------------+
|$ E' T' ) E' T'          |/*10) $                      |                         |                         |
+-------------------------+-------------------------+-------------------------+-------------------------+
```

结论

可以看出，程序能够对给定的文法生成正确的LL1分析表，并且对输入的串给出正确的分析结果和过程。

LR(1)

grammar2中存储了拓广文法

```
E'
E' → E
E → E + T | E - T | T
T → T * F | T / F | F
F → ( E ) | num
```

测试1

输入文件:
(3-2)*(4/10)

运行: ./lr1 grammar2 source > lr1.out 输出到文件中

30 sets.

No.	Elements
0	E -> .E+T,$ + - E -> .E-T,$ + - E -> .T,$ + - E' -> .E,$ F -> .(E),$ * + - / F -> .num,$ * + - / T -> .F,$ * + - / T -> .T*F,$ * + - / T -> .T/F,$ * + - /
1	E -> E.+T,$ + - E -> E.-T,$ + - E' -> E.,$
2	E -> T.,$ + - T -> T.*F,$ * + - / T -> T./F,$ * + - /
3	E -> E+.T,$ + - F -> .(E),$ * + - / F -> .num,$ * + - / T -> .F,$ * + - / T -> .T*F,$ * + - / T -> .T/F,$ * + - /
4	E -> E-.T,$ + - F -> .(E),$ * + - / F -> .num,$ * + - / T -> .F,$ * + - / T -> .T*F,$ * + - / T -> .T/F,$ * + - /
5	E -> E+T.,$ + - T -> T.*F,$ * + - / T -> T./F,$ * + - /
6	E -> E-T.,$ + - T -> T.*F,$ * + - / T -> T./F,$ * + - /
7	E -> .E+T,) + - E -> .E-T,) + - E -> .T,) + - F -> (.E),$ * + - / F -> .(E),) * + - / F -> .num,) * + - / T -> .F,) * + - / T -> .T*F,) * + - / T -> .T/F,) * + - /
8	E -> .E+T,) + - E -> .E-T,) + - E -> .T,) + - F -> (.E),) * + - / F -> .(E),) * + - / F -> .num,) * + - / T -> .F,) * + - / T -> .T*F,) * + - / T -> .T/F,) * + - /
9	E -> E.+T,) + - E -> E.-T,) + - F -> (E.),$ * + - /
10	E -> E.+T,) + - E -> E.-T,) + - F -> (E.)) * + - /
11	E -> T.,) + - T -> T.*F,) * + - / T -> T./F,) * + - /
12	E -> E+.T,) + - F -> .(E),) * + - / F -> .num,) * + - / T -> .F,) * + - / T -> .T*F,) * + - / T -> .T/F,) * + - /
13	E -> E-.T,) + - F -> .(E),) * + - / F -> .num,) * + - / T -> .F,) * + - / T -> .T*F,) * + - / T -> .T/F,) * + - /
14	E -> E+T.,) + - T -> T.*F,) * + - / T -> T./F,) * + - /
15	E -> E-T.,) + - T -> T.*F,) * + - / T -> T./F,) * + - /
16	F -> .(E),$ * + - / F -> .num,$ * + - / T -> T*.F,$ * + - /
17	F -> .(E),$ * + - / F -> .num,$ * + - / T -> T/.F,$ * + - /
18	F -> num.,$ * + - /
19	F -> (E).,$ * + - /
20	F -> .(E),) * + - / F -> .num,) * + - / T -> T*.F,) * + - /
21	F -> .(E),) * + - / F -> .num,) * + - / T -> T/.F,) * + - /
22	F -> num.,) * + - /
23	F -> (E).,) * + - /
24	T -> F.,$ * + - /
25	T -> T*F.,$ * + - /
26	T -> T/F.,$ * + - /
27	T -> F.,) * + - /
28	T -> T*F.,) * + - /
29	T -> T/F.,) * + - /

Analysis table:

	action							goto			
	$	(	)	*	+	-	/	num	E	E'	F	T
0		shift 7						shift 18	1		24	2
1	ACC				shift 3	shift 4						
2	reduce E -> T			shift 16	reduce E -> T	reduce E -> T	shift 17					
3		shift 7						shift 18			24	5
4		shift 7						shift 18			24	6
5	reduce E -> E+T			shift 16	reduce E -> E+T	reduce E -> E+T	shift 17					
6	reduce E -> E-T			shift 16	reduce E -> E-T	reduce E -> E-T	shift 17					
7		shift 8						shift 22	9		27	11
8		shift 8						shift 22	10		27	11
9			shift 19		shift 12	shift 13						
10			shift 23		shift 12	shift 13						
11			reduce E -> T	shift 20	reduce E -> T	reduce E -> T	shift 21					
12		shift 8						shift 22			27	14
13		shift 8						shift 22			27	15
14			reduce E -> E+T	shift 20	reduce E -> E+T	reduce E -> E+T	shift 21					
15			reduce E -> E-T	shift 20	reduce E -> E-T	reduce E -> E-T	shift 21					
16		shift 7						shift 18			25	
17		shift 7						shift 18			26	
18	reduce F -> num			reduce F -> num	reduce F -> num	reduce F -> num	reduce F -> num					
19	reduce F -> (E)			reduce F -> (E)	reduce F -> (E)	reduce F -> (E)	reduce F -> (E)					
20		shift 8						shift 22			28	
21		shift 8						shift 22			29	
22			reduce F -> num	reduce F -> num	reduce F -> num	reduce F -> num	reduce F -> num					
23			reduce F -> (E)	reduce F -> (E)	reduce F -> (E)	reduce F -> (E)	reduce F -> (E)					
24	reduce T -> F			reduce T -> F	reduce T -> F	reduce T -> F	reduce T -> F					
25	reduce T -> T*F			reduce T -> T*F	reduce T -> T*F	reduce T -> T*F	reduce T -> T*F					
26	reduce T -> T/F			reduce T -> T/F	reduce T -> T/F	reduce T -> T/F	reduce T -> T/F					
27			reduce T -> F	reduce T -> F	reduce T -> F	reduce T -> F	reduce T -> F					

```
+---+----------------------+----------------------+----------------------+----------------------+----------------------+----------------------+----------------------+----------------------+----------+----------+----------+----------+
|28 |                      |                      |reduce T -> T*F       |reduce T -> T*F       |reduce T -> T*F       |reduce T -> T*F       |reduce T -> T*F       |                      |          |          |          |          |
+---+----------------------+----------------------+----------------------+----------------------+----------------------+----------------------+----------------------+----------------------+----------+----------+----------+----------+
|29 |                      |                      |reduce T -> T/F       |reduce T -> T/F       |reduce T -> T/F       |reduce T -> T/F       |reduce T -> T/F       |                      |          |          |          |          |
+---+----------------------+----------------------+----------------------+----------------------+----------------------+----------------------+----------------------+----------------------+----------+----------+----------+----------+
LR(1) Parsing Complete.
Full Parsing Sequence:
+----------------------------------------+----------------------------------------+----------------------------------------+----------------------------------------+
|States                                  |Prefix                                  |Input                                   |Strategy                                |
+----------------------------------------+----------------------------------------+----------------------------------------+----------------------------------------+
|0                                       |                                        |( num - num ) * ( num / num ) $         |shift 7                                 |
+----------------------------------------+----------------------------------------+----------------------------------------+----------------------------------------+
|0 7                                     |(                                       |num - num ) * ( num / num ) $           |shift 22                                |
+----------------------------------------+----------------------------------------+----------------------------------------+----------------------------------------+
```

States	Prefix	Input	Strategy
0		(num - num) * (num / num) $	shift 7
0 7	(	num - num) * (num / num) $	shift 22
0 7 22	(num	- num) * (num / num) $	reduce F -> num
0 7 27	(F	- num) * (num / num) $	reduce T -> F
0 7 11	(T	- num) * (num / num) $	reduce E -> T
0 7 9	(E	- num) * (num / num) $	shift 13
0 7 9 13	(E -	num) * (num / num) $	shift 22
0 7 9 13 22	(E - num	) * (num / num) $	reduce F -> num
0 7 9 13 27	(E - F	) * (num / num) $	reduce T -> F
0 7 9 13 15	(E - T	) * (num / num) $	reduce E -> E-T
0 7 9	(E	) * (num / num) $	shift 19
0 7 9 19	(E)	* (num / num) $	reduce F -> (E)
0 24	F	* (num / num) $	reduce T -> F
0 2	T	* (num / num) $	shift 16
0 2 16	T *	(num / num) $	shift 7
0 2 16 7	T * (	num / num) $	shift 22
0 2 16 7 22	T * (num	/ num) $	reduce F -> num
0 2 16 7 27	T * (F	/ num) $	reduce T -> F
0 2 16 7 11	T * (T	/ num) $	shift 21
0 2 16 7 11 21	T * (T /	num) $	shift 22
0 2 16 7 11 21 22	T * (T / num	) $	reduce F -> num
0 2 16 7 11 21 29	T * (T / F	) $	reduce T -> T/F
0 2 16 7 11	T * (T	) $	reduce E -> T
0 2 16 7 9	T * (E	) $	shift 19
0 2 16 7 9 19	T * (E)	$	reduce F -> (E)
0 2 16 25	T * F	$	reduce T -> T*F
0 2	T	$	reduce E -> T
0 1	E	$	ACC

测试2

输入文件：
(3-2)*(4/*10)

运行： ./lr1 grammar2 source > lr1.out 输出到文件中
（省略文法部分）

```
LR(1) Parsing Failed.
Partial Parsing Sequence:
+------------------------+------------------------+------------------------+------------------------+
|States                  |Prefix                  |Input                   |Strategy                |
+------------------------+------------------------+------------------------+------------------------+
|0                       |                        |( num - num ) * ( num /*10) $|shift 7                 |
+------------------------+------------------------+------------------------+------------------------+
|0 7                     |(                       |num - num ) * ( num /*10) $|shift 22                |
+------------------------+------------------------+------------------------+------------------------+
|0 7 22                  |( num                   |- num ) * ( num /*10) $ |reduce F -> num         |
+------------------------+------------------------+------------------------+------------------------+
|0 7 27                  |( F                     |- num ) * ( num /*10) $ |reduce T -> F           |
+------------------------+------------------------+------------------------+------------------------+
|0 7 11                  |( T                     |- num ) * ( num /*10) $ |reduce E -> T           |
+------------------------+------------------------+------------------------+------------------------+
|0 7 9                   |( E                     |- num ) * ( num /*10) $ |shift 13                |
+------------------------+------------------------+------------------------+------------------------+
|0 7 9 13                |( E -                   |num ) * ( num /*10) $   |shift 22                |
+------------------------+------------------------+------------------------+------------------------+
|0 7 9 13 22             |( E - num               |) * ( num /*10) $       |reduce F -> num         |
+------------------------+------------------------+------------------------+------------------------+
|0 7 9 13 27             |( E - F                 |) * ( num /*10) $       |reduce T -> F           |
+------------------------+------------------------+------------------------+------------------------+
|0 7 9 13 15             |( E - T                 |) * ( num /*10) $       |reduce E -> E-T         |
+------------------------+------------------------+------------------------+------------------------+
|0 7 9                   |( E                     |) * ( num /*10) $       |shift 19                |
+------------------------+------------------------+------------------------+------------------------+
|0 7 9 19                |( E )                   |* ( num /*10) $         |reduce F -> (E)         |
+------------------------+------------------------+------------------------+------------------------+
|0 24                    |F                       |* ( num /*10) $         |reduce T -> F           |
+------------------------+------------------------+------------------------+------------------------+
|0 2                     |T                       |* ( num /*10) $         |shift 16                |
+------------------------+------------------------+------------------------+------------------------+
|0 2 16                  |T *                     |( num /*10) $           |shift 7                 |
+------------------------+------------------------+------------------------+------------------------+
|0 2 16 7                |T * (                   |num /*10) $             |shift 22                |
+------------------------+------------------------+------------------------+------------------------+
```

结论

可以看出，程序能够对给定的文法生成正确的LR1分析表，并且对输入的串给出正确的分析结果和过程。

Yacc

`in`中内容为：

```
(11+2)+(3/4)
```

调用`./cc in`，得到输出的序列：

```
mark = '('
num = 11
mark = '+'
num = 2
mark = ')'
mark = '+'
mark = '('
num = 3
mark = '/'
num = 4
mark = ')'
mark = '\n'
success
```

在`in`中写一个不合法的表达式：

```
(11+2)+(3.3/4*2
```

调用`./cc in`，得到输出序列：

```
mark = '('
num = 11
mark = '+'
num = 2
mark = ')'
mark = '+'
mark = '('
num = 3.3
mark = '/'
num = 4
mark = '*'
num = 2
mark = '\n'
error syntax error
```